1 Scheme

1.1 What would Scheme do?

scm> (and 0 2 200)

scm> (or True (/ 1 0))

scm> (and False (/ 1 0))

scm> (not 3)

1.2 What would Scheme display?

scm> (define a (+ 1 2))

scm> a

scm> (define b (+ (* 3 3) (* 4 4)))

scm> (+ a b)

scm> (= (modulo 10 3) (quotient 5 3))

scm> (even? (+ (- (* 5 4) 3) 2))

scm> (if (and #t (/ 1 0)) 1 (/ 1 0))

scm> (if (> (+ 2 3) 5) (+ 1 2 3 4) (+ 3 4 (* 3 2)))

scm> ((if (< 9 3) + -) 4 100)

scm> (if 0 #t #f)
1.3 Write two Scheme expressions that are equivalent to the following Python statement
   - one defining a function directly, and the other creating an anonymous lambda that
     is then bound to the name cat:
     
     ```python
     cat = lambda meow, purr: meow + purr
     ```

1.4 Spot the bug(s). Test out the code and your fixes in the scheme interpreter!
   (https://scheme.cs61a.org/)

   ```scheme
   (define (sum-every-other lst)
     (cond ((null? lst) lst)
           (else (+ (cdr lst)
                     (sum-every-other (caar lst)) ))))
   ```

1.5 Define **sixty-ones**, a function that takes in a list and returns the number of times
   that 1 follows 6 in the list.

   ```scheme
   > (sixty-ones '(4 6 1 6 0 1))
   1
   > (sixty-ones '(1 6 1 4 6 1 6 0 1))
   2
   > (sixty-ones '(6 1 6 1 4 6 1 6 0 1))
   3
   ```

1.6 Define **no-elevens**, a function that takes in a number n, and returns a list of all
   distinct length-n lists of 1s and 6s that do not contain two consecutive 1s.

   ```scheme
   > (no-elevens 2)
   ((6 6) (6 1) (1 6))
   > (no-elevens 3)
   ((6 6 6) (6 6 1) (6 1 6) (1 6 6) (1 6 1))
   > (no-elevens 4)
   ((6 6 6 6) (6 6 6 1) (6 6 1 6) (6 1 6 6) (6 1 6 1) (1 6 6 6) (1 6 6 1) (1 6 1 6))
   ```
1.7 Define `remember`, a function that takes in another zero-argument function `f`, and returns another function `g`. When called for the first time, `g` will call `f` and pass on its return value. When called subsequent times, `g` will remember its previous return value and return it directly, without calling `f` again.

(Hint: look up `set!` in the Scheme spec!)

```scheme
(define (remember f)
  )
```

```scheme
scm> (define (f) (print "hello!") 5)
scm> (define g (remember f))
scm> (f)
hello!
5
scm> (g)
hello!
5
scm> (g)
5
```

**Check your understanding**

- How are call expressions (like `(+ 1 2 3)`) evaluated? What about special forms, like `(or #f #t (/ 1 0))`?
- What is the purpose of the `quote` special form?
2 Scheme Lists

2.1 Draw out a box-and-pointer diagram for the following list:

```scheme
(define nested-lst (list 1 (cons 2 (cons 3 'nil)) '(4 5 6) 7))
nested-lst
```

Then, write out what Scheme would display for the following expressions:

```scheme
(scm> (cdr nested-lst))
(scm> (cdr (car (cdr nested-lst))))
(scm> (cons (car nested-list) (car (cdr (cdr nested-list)))))
```
**Extra**

2.2 Notice that the builtin `append` takes in, not a list of lists, but an arbitrary number of lists as arguments, which it then concatenates together. Implement `better-append`, which behaves in such a manner, allowing the caller to pass in an arbitrary number of arguments. You may use `concat` from the previous question.

(Hint: look up “variadic functions” in the Scheme spec!)

```
scm> (better-append '(1 2 3))
(1 2 3)
scm> (better-append '(1 2 3) '(2 3 4))
(1 2 3 2 3 4)
scm> (better-append '(1 2 3) '(2 3 4) '(3 4 5))
(1 2 3 2 3 4 3 4 5)
```

**Check your understanding**

- How can you get the third element of a Scheme list? Draw out a box-and-pointer diagram if you aren’t sure.

- What is the difference between `eq?` and `equal?` in the context of Scheme lists? Construct two lists `lst1` and `lst2` such that `(equal? lst1 lst2)` is #t but `(eq? lst1 lst2)` is #f.